**EECS2011E**

**Assignment 2 Report**

**Modeling of a Deterministic CPU scheduling algorithm**

**Modeling of a Deterministic CPU scheduling algorithm**

Simulating a CPU scheduling service is not an easy task. In this assignment we attempt to build a dynamic CPU scheduling system. It is dynamic in that the program has no hard-coded processes or all processes are randomly generated just as an actual system generates many diverse processes with varying amounts of CPU and IO time requirements.

In this assignment a process is defined as filling an array with 'characters'. A Process class has the method run(); when this method is called once it adds a character to the ArrayList of the process - processSite, i.e. processSite will add one character element to its set.

When a process has finished its temporary CPU bound process (i.e. the filling of the ArrayList) it is transfered to the an IO queue. The process spends a designated amount of time in the IO queue (i.e. the class Device). The time that the process spends in the IO is specified by the Process.BurstSpec inner class. After it is done with the IO wait time, the process returns back to the ready queue of the current scheduler.

**Choice of Data Structures:**

The data structure that is used frequently is a Queue, specifically the java.util.concurrent.LinkedBlockingQueue class. It has great functionality that allows us to use enquing and dequing methods with ease. Moreover, instead of array I have resorted to using ArrayLists since they are easily expandable.

**The Process class**

- The process class encapsulates the essential features of a process. Intuitively a process is an activity that takes time from start to finish. As mentioned above an instance of a Process class has list of Process.BurstSpec instances that each specify a CPU burst time, an IO burst time, and the device number associated with the IO burst. For example a a list of Process.BurstSpec can be like this:

**Example 1**: pre-determined CPU and IO burst times for a process

new BurstSpec(6,10,0),

new BurstSpec(7,3,0),

new BurstSpec(4,2,0)

The first of the above array of BurstSpec inner classes specifies that when this particular process runs it must have 6 seconds of CPU burst time followed by 10 seconds of IO burst time in device number 0;

Other attributes of the Process class are:

arrivalTime:

It is set by the process generator class (ProcessGen) at the time when this process is transferred to the ready queue of a scheduling service.

cpuBurst:

It is the total amount of time the process spends in the CPU. For example, consider the above pre-determined specification in Example 1; the total CPU burst time will be 6 + 7 + 4 = 17 quanta (time slices). However, in an attempt to make this assignment a dynamic simulator, the CPU and IO burst times are not calculated automatically by the class. The determination of the CPU and IO burst times are discussed later.

ioBurst:

It is the total amount of time the process spends in the IO device queue. Similar to the calculation of the CPU burst time above, the IO burst time is calculated by adding the IO specification in the list of BurstSpec inner-classes. Taking Example 1, we would have ioBurst = 10 + 3 + 2 = 15.

processPriority:

is the priority assigned to a process upon creation. This value is used only in a priority scheduling service. Low values are for higher priorities.

processSite:

is an ArrayList that is filled by one character when the process is executed for one CPU time unit.

processID:

is the ID of a process assigned to it at the time of creation.

**Generation of Processes**

- Processes are generated automatically by programs running in the computer. At any particular time the scheduling algorithm of the operating system loads a process into the CPU. When the current process requests for an IO activity this process is removed from the CPU and is transferred to the IO queue, then, upon completion of its wait time it is taken back to the ready queue.

- To emulate this automatic process, we have used the class ProcessGen. The ProcessGen class is a singleton class that runs its own thread upon instantiation. This class can wait a designated amount of time and it creates a new Process class and enqueues it to a temporary field named 'jobQueue'. Then it transfers the Processes in the jobQueue to the ready queue found in the currently used schedueling service(FCFS, SJF,...). The thread stops after adding a certain number of processes into the ready queue.

**FCFS**

- If the a process in a FCFS schedule has both a CPU burst and IO burst times then when the IO burst time for this particular process takes place, the CPU would stay idle until this same process is returned back to it. If the IO burst for this process is 6 seconds the CPU will lose 6 valuable seconds of processing. To Make the FIFO efficient, the second process following the current one will access the CPU when the first process goes into IO. When the first goes into IO device queue, the second will take over CPU and takeover the process until it completes all processing or until it leaves CPU and goes into IO queue.

- The implementation of the FCFS scheduler is said to be among the simplest scheduling algorithms. It involves processing the first element until completion and moving on to the next. Different processes have their own CPU and IO time specifications. So if the first process is on its IO burst, the CPU will be left idle with nothing to process until the currently running process finishes its IO burst and returns to the CPU for further processing. This results in an inefficient usage of the CPU. Therefore, when a currently running process goes into its IO waiting phase the next process will take over and the first process will be processes after the second is done or until it passes into its IO waiting phase. Implementing the FCFS in this way reduces the idling time of the CPU but it doesn't completely prevent idling.

**FCFS implementation**

- The FCFS class contains a readyQueue named 'list'. This readyQueue is a static LinkedBlockingQueue<Process> to prevent the creation of several ready queues; that is, a FCFS scheduler should only contain one ready queue.

- The heart of the FCFS is a method called the cpuProcess(). This method controls what gets processed and, when required, passing the process into the IO queue into a particular IO device.

Fields of FCFS class:

private static LinkedblockingQueue<Process> list: it is the FCFS queue and the poll() function is used to take the foremost item from the queue and process it until the process requests for I/O service. When a process finishes its IO wait time it is returned enqueued back at the tail of this queue.

private ArrayList<Process> copyList: this ArrayList stores a reference of all the processes that are processed in the FCFS. Because the above list loses all reference to all processes after they are processed. The copyList is used in the end to print out the process detail at the end of all processes.

Methods of FCFS class:

Getter methods: they return a reference of the fields of the class.

Setter methods:

void setList() : it sets the reference of the list field.

Operation methods:

**void makeCopy()** : it stores the reference of all unique processes passed into the class

void cpuProcess() : this method acts like the CPU of the system. It contains an infinite while loop that only breaks out when all the processes in the 'list' field is emptied of all processes. In one iteration, first the head process is polled and passed into a a for loop that iterates as many times as specified by the processe's first CPU time requirement. To slow down the loop so it is easier to observe the process, there is a Utility.sleep() method called from the Utility class that sleeps the thread running the FCFS class by one second. Outside of the for loop there are methods Process.upCpuBurst() that record the time spent by the CPU running the process. Once this loop is finished, there is an if statement that sets up the process for the IO queue. If the process requires an IO time this process is completely passed out of the FCFS into the Device class by means of the Device constructor. The Device constructor also takes in the reference of the 'list' field so that when the IO wait time is done it can be returned back to the FCFS class. If the process doesn't have any IO requirement (i.e. the CPU/IO specification or BurstSpec has an IO of 0) the process is either enqueued back into the ready queue (list field) or if there are no specifications left the process is removed and terminated. There is an else statement that is used to specifies the waiting time of the system. What this means is is the CPU will wait for a process for the specified amount of time (in this case 10 sec) and if no process in enqueued in the ready queue (list field) the system shuts down and calls the FCFS.calculate() method. However, if there is another process the infinite while loop keeps on going until all processes are exhausted.

void calculate() : this method calculates the average turnaround time and the average waiting time for all the processes in the ready queue of the FCFS class. It also prints all the details of the processes including the time the process is created, the time that there is a state change, the time when the processes was running, the time the process is in IO queue and so on. All this information is stored inside a ProcessRecord class.

**Device class**

The Device class is implemented to serve as an IO service class. The Device class is a multiton class that creates a device with a specified id number only once. If the same device is requested again by another process the device reference is called from the multiton list of devices. One special feature of this class is that it runs on a separate thread than the CPU scheduling classes. For example if a process in the FCFS class calls for device number 1, the Device class creates Device-1 or if Device-1 is already created gets Device-1 from the Device class's HashMap and have the process stay in the Device-1 queue (deviceQueue field) for as long as the process needs to stay. The fact that it runs on a separate thread from the scheduling classes allows the scheduler to move to the next process. This is a very important feature of this implementation and so I will describe it in detail below.

For example, if the FCFS scheduler and the Device class run on the same thread then their execution is necessarily sequential. A CPU scheduler that waits for a process to finish IO then proceeds to the next process wastes valuable time of processing and will be inefficient. In this implementation when a process is passed into the Device class for its IO wait requirement, it is essentially non-existent from the ready queue of the scheduler (FCFS) and most importantly the Device classes' process is a separate thread and doesn't make the FCFS processes to wait for the process. Therefore, the FCFS can proceed to the next process without waiting for the one passed into the Device class! When the process finishes its IO wait time, it goes back into the scheduler's ready queue. This implementation is very efficient in that it removes any CPU idling time by separating the CPU processes and IO processes!

Device Implementation:

Fields:

deviceQueue: it is a LinkedBlockingQueue<Process> that stores all processes that use this particular device. Each device created has its own deviceQueue that store processes. This data structure allows for a first-come-first-serve processing of all processes.

returnQueue: it is a static reference passed from the scheduling class (i.e. FCFS class). It is used to return a process that finished its IO wait time if the process has remaining CPU processing. If a process has no CPU requirement time it is not returned into this queue.

currentDevices: it is a HashMap<Integer, Device> that stores and integer key as the device id and a Device instance. Whenever an existing device is requested it is fetched out from this HashMap. If a new device is created it is stored in this map too.

executorService: it is an ExecutorService class that generates a new Device thread that is separate from the CPU scheduler's thread and the threads of all other Device threads.

Methods:

Device getDevice(int,Process,LinkedBlockingQueue<Process>) : this method takes in an int device id, a process and a return LinkedBlockingQueue of the scheduler and then constructs a new Device if the device id is not found in the currentDecvices HashMap. It then sets the returnQueue fied to the passed queue reference. Once the Device is retrieved, the reference of it is returned to the caller.

void deviceEnqueue(Process) : this method is used to pass the reference of the process that requires IO into the specified device. to prevent redundant reference passing it contains an if statement that blocks the parameter from being passed again it already exists in the deviceQueue.

void ioService() : this is the heart of the IO device. it makes the current process to wait in IO for the specified amount of time. It contains a for loop that counts for x amount of IO time with a temporary 1 second thread sleep. It also records the IO wait time for the process by the Process.upIoBurst(Long) method. Once the IO wait is done it removes the block of BurstSpec class that contains the IO time. The next block of code checks if there is any CPU/IO requirements (BurstSpec) left; if there is left then the process will be enqueued back into the ready queue of the scheduler and if it is not then the termination time of the process will be set and removed from the deviceQueue effectively losing the reference of the process. If the process is not enqueued back into the ready queue of the scheduler it will be lost or essentially terminated!

void run() : the run method is an overridden method of the Runnable interface the Device class implements. It runs the ioService() method inside an infinite while loop as long as the device queue isnot empty.

void timeThread() : this method starts the Device instance's thread. The thread doesn't end because the while loop in the run() method prevents the thread from ending. This feature is important because once the device is started it will stay live as long as the system CPU is running. The device might be required again at some time in the future and thus with the device being live the scheduler can just call its reference and pass in the process that needs IO it this device.

How the Main, FCFS, ProcessGen, Device, Process and Utility classes work together:

Below I will explain how the 5 classes above work together to execute a process. The Main class starts the ProcessGen singleton class thread, the FCFS class and pass in a common LinkedBlockingQueue<Process> into both classes via the respective setter methods. The ProcessGen class runs concurrently along the FCFS class. The FCFS initially has no processes to schedule and so it waits for 10 seconds after startup. The ProcessGen class has the capability to generate new processes and pass them into the FCFS via the shared reference they both have.

First let us make a process with predetermined CPU and IO time requirements:

P1: new BurstSpec(6,10,0),

new BurstSpec(7,3,0)

P2: new BurstSpec(6,10,0),

new BurstSpec(7,3,0)

The Process P will store these time requirement specifications in the timeSpec LinkedList. The process will have its creationTime, arrivalTime, priority set by the ProcessGen jobProcessEnqueue() method. And the multiProcessEnqueu() method passes the process into the FCFS shared queue.

As soon as the FCFS class gets the process P1 passed into its queue it will start the cpuProcess(). The cpuProcess() method takes the CPU time from BurstSpec(6,10,0), which is 6, and runs the for loop 6 times. In between each iteration the Utility.sleep() method is called and a 1 second delay is effected to slow down the loop. The P1.upCpuBurst(-Utility.time()) and P1.upCpuBurst(Utility.time()) bouding the loop pass in the initial time and final time of the loop and the difference of the times is added as the CPU burst time in the cpuBurst field. Once the CPU time is processed P1 requires an IO time of 10 seconds. For this a Device class is constructed via the Device.getDevice(0, P1, readyQueue) and the device number in the BurstSpec- the third number, 0 - along with the current Process, P1 and a return address - i.e. the ready queue of FCFS - named 'list' - are passed.

This is a branching point of the FCFS thread, which is under the Main.main() method's thread, and the newly created Device-0 thread. Once P1 is passed into Device-0 the device thread runs and the Device.ioService() method is called. This method takes the IO requirement specified by the P1 BurstSpec, which is 10, and runs the loop inside it 10 times with a 1 second sleep time similar to the FCFS.cpuProcess inner loop. IO time spent in the loop is recorded by P1.upIoBurst(-Utility.time()) for initial time and P1.upIoBurst(UItility.time()) for final time and their difference is computed and stored in the ioBurst field of P1. The BurstSpec that contains (6,10,0) will be removed since both the CPU and IO requirements are met so it will be removed. P1 will be enqueued back into the returnQueue, which is the ready queue of the FCFS scheduler.

While the Device-0 class is running FCFS runs in its own thread too. P1 is polled from the ready queue and passed into the Device-0 queue, when the while loop of the FCFS reiterates it will find P2 and carry out the CPU time requirement of P2 which is 6 seconds just like P1. Both P1 's IO wait and P2's CPU process are concurrent activities in different threads. In cases where P2 finishes its CPU time and P1 didn't finish its IO wait time P2 will be enqueued in Device-0 deviceQueue and IO times will be processed in a first-come-first-served manner. All processes return between the FCFS and Device classes until they are done and finally the process ends and system shuts down. At the end of the all processes the process detail is printed and the average waiting time and average turnaround time.

Every time a process is created and creation time is assigned, arrival time is assigned, termination time is assigned, a process runs in CPU, a process is in IO a Process.record() method is called inside the current process. The record() method creates a ProcessRecord class that encapsulates key information about the process and stores this ProcessRecord class inside an ArrayList. This ArrayList is used as an evidence of the process.

**Calculation of Waiting time and Turnaround time**

Waiting time is the time a process spends in CPU ready queue or IO device queue.

Turnaround time is the time a process takes from creation to end of execution.

Turnaround time for a single process is the termination time minus creation time. Therefore,

turnaroundTime = P1.getTerminationTime() - P1.getCreationTime().

Waiting time for a single process is the turnaround time minus the cpu burst time. Thus,

waitingTime = turnaroundTime - P1.getCpuBurst().

The average time is, simply adding all the turnaround and waiting times for a processes and dividing by the number of processes. The calculations are done in the FCFS.calculate() method.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

All other scheduling algorithms implemented are a derivative of the FCFS algorithm. The main difference they have from the FCFS is the different way their cpuProcess() method is implemented.

1. Non-Premptive First Come First Served.

2. Preemptive Round-Robin (RR) Scheduling

3. Nonpreemptive Shortest-Job-First (SJF).

4. Preemptive SJF (Shortest-Remaining-Time-First).

5. Nonpreemptive Priority Scheduling.

6. Preemptive Priority Scheduling

7. Multilevel Queue Scheduling

8. Multilevel Feedback Queue Scheduling

**2. Preemptive Round-Robin (RR) Scheduling**

The Preemptive Round-Robin (referred to as RR)'s difference from the FCFS is that it has a certain time-slice restriction. The time restriction is set in the RR.runLimit field. The runLimit is set to 2 in the RR constructor. The first process of the ready queue is taken, runs for 2 time-slices (2 seconds) and is preempted by the second process in the ready queue. If during the 2 time slices the first process finishes its work it goes into IO and come back to the end of the ready queue. Basically, every process runs for 2 time slices until they finish their task and the process ends. Finally, the calculate() method is called and it displays the calculated results.

**3. Nonpreemptive Shortest-Job-First (SJF)**

Similar to the FCFS, processes are generated and passed into the ready queue of the Non-Preemptive Shortest Job First (refered to as SJF) by the ProcessGen thread. Once processes are enqueued in the ready queue (list field) they can be processed. The idea behind this implementation is that the process with the shortest current CPU burst to take precedence. The first process enqueued will take the CPU (run in the cpuProcess() loop) until it leaves the CPU for IO. At this time, the sjfSorter(LinkedBlockingQueue<Process>) method is called and it uses a locally implemented Comparator interface with an overridden int compare(Process, Process) to sort out the process with the smallest current CPU time. If two processes have equal times then they will be compared by arrivalTime. What it means by 'current CPU time' is the Process may have several BurstSpec (CPU/IO specifications in order) and the Comparator will compare based on the first occurring BurstSpec. By using the sjfSorter(...) method a process is fetched from the ready queue and passed into the for loop of the cpuProcess() method. The rest of the process is similar to the FCFS.

**4. Preemptive SJF (Shortest-Remaining-Time-First).**

The Preemptive SJF (refered to as pSJF) is a composition of the non-preemptive SJF and the RR. The first process of the ready queue is processed for a certain time-slice (similar to the RR) and then a new process is chosen in a similar fashion to the non-preemptive SJF by means of the sjfSorter() method. The time slice is set to enable the scheduling algorithm to check if there is any other process that has a smaller CPU run time than the current one. The newly chosen process is also given the same time slice and another session of checking for a short running process is done. This process iterates until all processes are done.

**5. Nonpreemptive Priority Scheduling.**

The Non preemptive Priority Scheduling is the same as the SJF except that processes are compared not by shortest CPU run time but by the priority assigned to them when they are created in the ProcessGen thread.

**6. Preemptive Priority Scheduling**

The Preemptive Priority Scheduling is the same as the Preemptive SJF (Shortest-Remaining-Time-First) except that process are compared by priority. It has all the functionality of the Preemptive SJF.

**7. Multilevel Queue Scheduling**

The idea behind a Multilevel Queue Scheduling (referred to as MQS) is to have more than one queues that accept their own set of processes but the foreground queue of processes will have absolute priority over the background queue. To implement this, the cpuProcessXX() method from he RR and FCFS classes were taken and modified to run only once. the cpuProcess() method of this scheduler calls the cpuProcessXX() methods of the RR and FCFS schedulers. By using if statement priority is given to the RR obtained cpuProcessXX() method so all the processes in the ready queue prepared for the RR scheduler are processed first.

**8. Multilevel Feedback Queue Scheduling**

This scheduler has the same concept as the Multilevel Queue Scheduler. However, the difference between them is that the first one has queues that are fed processes separately. In the case of this scheduler, even if it has multiple queues the processes are fed through the first and first priority queue. It also contains cases where a process could be promoted to a high priority queue or demoted to a low priority queue based on its CPU burst time. If a process that is enqueued into the first high priority queue doesn't finish its execution within the time-slice specified it will be demoted to a lower priority queue and the next process will take the CPU. When all the processes in the high priority queue are exhausted by demotion or termination, the next highest priority queue will take over and it will provide a greater time-slice than the first one. Interestingly, in this implementation processes in the lower priority queues are promoted up a notch when they come back from their IO wait time. In addition, a process in a lower queue is promoted up if it has a CPU time less than the ready queue above it. The three ready queues in this scheduler in decreasing priority are RoundRobin with a time slice of 5 seconds, Round-Robin with a time slice of 9 seconds and the least one, an FCFS ready queue.

**How testing was done**

This program dynamically produces all processes and automatically runs the schedulers. To test if the final results (Turnaround time and Waiting time) were indeed accurate, the program was made to print a summary of the Process classes generated at the end of its run. Therefore, I was able to calculate the actual average turnaround time and average waiting time of the entire run. In addition the context switch time was assumed to be 1 second for each scheduler. Also, as seen in the Test\_output.txt file the details of all processes are printed out. By following this step by step I was able to confirm the accuracy of the program. For every run the output.txt file will store the details of the each run.

Summary of the Scheduler

Process: 1 Creation: 0 Arrival: 3,Termination: 40 Priority: 10 , State: 4, Arr: [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O], BS: CP: 5, IO: 3, DN: 2 -- CP: 3, IO: 3, DN: 3 -- CP: 7, IO: 2, DN: 2 --

Process: 2 Creation: 1 Arrival: 7,Termination: 44 Priority: 7 , State: 4, Arr: [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R], BS: CP: 8, IO: 5, DN: 2 -- CP: 7, IO: 5, DN: 3 -- CP: 3, IO: 2, DN: 2 --

Avg Waiting time: 24.50

Avg Turnaround time: 41.50

Avg CPU burst time: 17.00

Avg IO burst time: 10.50

The expected calculation can be obtained from the summary, the first process started at time 0 and finished at time 40 seconds (including wait time and context switch time) and the second process started at time 1 and finished at time 44; the average turnaround time is (40+43)/2 = 41.5. The average CPU time is ((5 + 3 + 7) + (8 + 7 + 3))/2 = 16.5 (there is an error of 0.5). The average wait time is thus, 41.5 – 16.5 = 25 (an error of 0.5).

The other testing alternative is to draw out the Ghant diagrams for the given processes based on the output BurstSpec data (BS). Then determine the wait time and turnaround time from the diagram.

**General explanation on Asymptotic runs times of methods in the project:**

The asymptotic run time for each method is written in the javadoc description over the methods. Here I will summarize the big-oh run time form most methods.

1. All setters and getters are O(1)

2. Process class: all methods run O(1)

3. Device class: most methods run O(1)

the ioService() method runs O(n)

the run() method runs infinitly until system shutdown because of an infinite while loop

4. FCFS class and similar classes: most methods runs O(1)

the cpuProcess() method runs O(n^2), classes that share similar implementation of the

cpuProcess() class also run O(n^2) - (RR, P\_SJF, NP\_SJF,P\_Priority, NP\_Priority)

5. sjfSorter() method in P\_SJF, NP\_SJF,P\_Priority, NP\_Priority classes runs O(n)

6. cpuProcess() method in Multilevel, ML\_Feedback classes runs O(n^2)

7. Utility class: calculate() runs O(n) and all others run O(1)

**Difficuties faced on the Design and Implementation of this program**

Designing a CPU scheduling simulator is a very difficult task. One of the major difficulty was managing an efficient scheduler that minimizes CPU idling time. At first I designed and implemented a FCFS scheduler that does CPU process and then follow it with an IO wait, however during that time the CPU was idle. To improve this feature I had to research on how to run the FCFS scheduler and the Device concurrently. Even so I found it very difficult to close the threads once started.

Another difficult part is simulating the arrival of new processes into an already running CPU. The idea behind this comes from considering how a real computer works. The computer's CPU may have nothing to do but at any given time a process may arrive and it will be processed. To solve this problem, I implemented a ProcessGen class that automatically inserts new processes into a shared LinkedBlockingList<Process> field. This field is shared between the FCFS scheduler and the ProcessGen thread. Interestingly, Since the ProcessGen thread is a third concurrent thread, I was able to insert new processes into the ready queue without affecting the activities of the scheduler.

As you can see I have implemented a very elaborate and complicated project that has all scheduling algorithms. You may ask why I did this. The answer is a reading mistake. I thought I had to implement all the scheduling algorithms; however, it is later on after I finished all algorithms that I found out only one of them was required. Though, I can't deny that this is a great experience.